

Design of a Sliding Window over Asynchronous Event Streams

Yiling Yang^{1,2}, Yu Huang^{1,2*}, Jiannong Cao³, Xiaoxing Ma^{1,2}, Jian Lu^{1,2}

¹State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210093, China

²Department of Computer Science and Technology
Nanjing University, Nanjing 210093, China

csylyang@smail.nju.edu.cn, {yuhuang, xxm, lj}@nju.edu.cn

³Internet and Mobile Computing Lab, Department of Computing
Hong Kong Polytechnic University, Hong Kong, China
csjcao@comp.polyu.edu.hk



Abstract

The proliferation of sensing and monitoring applications motivates adoption of the event stream model of computation. Though sliding windows are widely used to facilitate effective event stream processing, it is greatly challenged when the event sources are distributed and asynchronous. To address this challenge, we first show that the snapshots of the asynchronous event streams within the sliding window form a convex distributive lattice (denoted by *Lat-Win*). Then we propose an algorithm to maintain *Lat-Win* at runtime. The *Lat-Win* maintenance algorithm is implemented and evaluated on the open-source context-aware middleware we developed. The evaluation results first show the necessity of adopting sliding windows over asynchronous event streams. Then they show the performance of detecting specified predicates within *Lat-Win*, even when faced with dynamic changes in the computing environment.

1 INTRODUCTION

Sensing devices such as wireless sensor motes and RFID readers are gaining adoption on an increasing scale for tracking and monitoring purposes. An emerging class of applications includes context-aware computing in a smart home/office [1], [2], supply chain management [3], and facility management [4]. These applications require the online processing of a large amount of events from multiple event sources, which necessitate the *event stream model* of computation [5], [4].

In tracking and monitoring applications, event streams are often generated from multiple distributed sources. More importantly, the event sources may not have a global clock or shared

*Corresponding author.

memory. Communications among the event sources may suffer from finite but arbitrary delay. It is a critical challenge how to process such *asynchronous event streams* [5], [1], [2].

For example in a smart office scenario, the context-aware middleware may receive the event stream of user's location updates from his mobile phone (we assume that the user's location can be decided by the access point his phone connects to) [1], [2]. The middleware may also receive event streams from sensors in the meeting room about whether there is a presentation going on. Due to the asynchrony among the event sources, the middleware cannot easily decide the composite global event "the user is in the meeting room, where a presentation is going on", in order to mute the mobile phone intelligently.

Coping with the asynchrony has been widely studied in distributed computing [6], [7]. One important approach relies on the "happen-before" relation resulting from message passing [8]. Based on this relation, various types of logical clocks can be devised [7], [9]. Based on logical time, one key notion in an asynchronous system is that all meaningful observations or global snapshots of the system form a distributive lattice [6], [7].

In tracking and monitoring applications, the events may quickly accumulate to a huge volume, and so will the lattice of global snapshots of the asynchronous event streams [7], [10]. Processing of the entire event streams is often infeasible and, more importantly, not necessary [11]. In such applications, we are often concerned only on the most recent events. This can be effectively captured by the notion of a *sliding window* [12], [5], [11]. Processing events within the window (discarding the stale events) can greatly reduce the processing cost. Also in the smart office scenario, user's location half an hour ago is often of little help in meeting his current need. Thus we can keep a sliding window (say latest 5 location updates) over the user's location stream.

Challenge of the asynchrony and effectiveness of the sliding window motivate us to study the following problem. In a system of n asynchronous event streams and one sliding window on each stream, we define an *n -dimensional sliding window* as the Cartesian product of the window on every event source. Considering the system of asynchronous event streams within the *n -dimensional sliding window*, does the lattice structure of global snapshots preserve? If it does, how to effectively maintain this lattice of snapshots at runtime? How to support effective detection of predicates over event streams within the window? Toward these problems, the contribution of this work is two-fold:

- We first prove that global snapshots of asynchronous event streams within the *n -dimensional sliding window* form a distributive lattice (denoted by *Lat-Win*). We also find that *Lat-Win* is a convex sub-lattice of the "original lattice" (obtained when no sliding window is imposed and the entire streams are processed);
- Then we characterize how *Lat-Win* evolves when the window slides over the asynchronous event streams. Based on the theoretical characterization, we propose an online algorithm to maintain *Lat-Win* at runtime.

A case study of a smart office scenario is conducted to demonstrate how our proposed *Lat-Win* facilitates context-awareness in asynchronous pervasive computing scenarios [1], [2]. The *Lat-Win* maintenance algorithm is implemented and evaluated over MIPA – the open-source context-aware middleware we developed [13], [14], [2]. The performance measurements first

show the necessity of adopting the sliding window over asynchronous event streams. Then the measurements show that using the sliding window, fairly accurate predicate detection (accuracy up to 95%) can be achieved, while the cost of event processing can be greatly reduced (to less than 1%).

The rest of this paper is organized as follows. Section 2 presents the preliminaries. Section 3 overviews how *Lat-Win* works, while Section 4 and 5 detail the theoretical characterization and algorithm design respectively. Section 6 presents the experimental evaluation. Section 7 reviews the related work. Finally, In Section 8, we conclude the paper and discuss the future work.

2 PRELIMINARIES

In this section, we first describe the system model of asynchronous event streams. Then we discuss the lattice of global snapshots of asynchronous event streams. Finally, we introduce the *n-dimensional sliding window* over asynchronous event streams. Notations used through out this work are listed in Table 1.

TABLE 1
Notations Used in Design of *Lat-Win*

Notation	Explanation
n	number of non-checker processes
$P^{(k)}, P_{che}$	non-checker / checker process ($1 \leq k \leq n$)
$e_i^{(k)}, s_i^{(k)}$	event / local state on $P^{(k)}$
$Que^{(k)}$	queue of local states from each $P^{(k)}$ on P_{che}
$W^{(k)}$	sliding window on a single event stream
$W_{min}^{(k)} / W_{max}^{(k)}$	the oldest/latest local state within $W^{(k)}$
w	uniform size of every $W^{(k)}$
W	n -dimensional sliding window over asynchronous event streams
\mathcal{G}	global state of the asynchronous event streams
\mathcal{C}	Consistent Global State (CGS)
$\mathcal{C}[k]$	k^{th} constituent local state of \mathcal{C}
LAT	original lattice of CGSs when no sliding window is used and the entire streams are processed
<i>Lat-Win</i>	lattice of CGSs within the n -dimensional sliding window
$\mathcal{C}_{min}, \mathcal{C}_{max}$	the minimal/maximal CGSs in <i>Lat-Win</i>

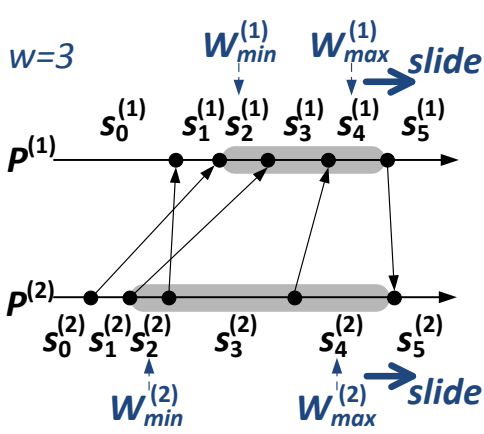
2.1 A System of Asynchronous Event Streams

In a tracking/monitoring application, we are faced with multiple distributed event sources which generate event streams at runtime. The event sources do not necessarily have global clocks or shared memory. The event sources are modeled as n *non-checker processes* $P^{(1)}, P^{(2)}, \dots, P^{(n)}$. Each $P^{(k)}$ produces a stream of *events* connected by its *local states*: “ $e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, e_2^{(k)}, \dots$ ”, as shown in Fig. 1(a). The event may be local, indicating status update of the entity being monitored and causing a local state change, or global, e.g. communication via sending/receiving messages. The non-checker processes form a loosely-coupled asynchronous system. We assume that no messages are lost, altered or spuriously introduced, as in [15], [16]. The underlying communication channel is not necessarily FIFO.

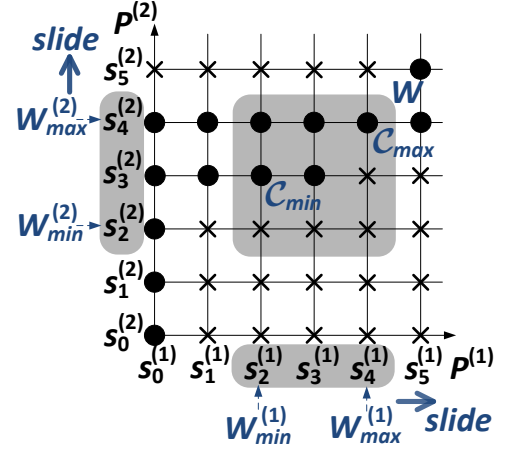
We re-interpret the notion of time based on Lamport's definition of the *happen-before* relation (denoted by ' \rightarrow ') resulting from message causality [8]. This happen-before relation can be effectively encoded and decoded based on the logical vector clock scheme [9]. Specifically, for two events $e_a^{(i)}$ and $e_b^{(j)}$ in the system of asynchronous event streams, we have $e_a^{(i)} \rightarrow e_b^{(j)}$ iff:

- $(i = j) \wedge (b = a + 1)$, or
- $(e_a^{(i)} = \text{send}(m)) \wedge (e_b^{(j)} = \text{receive}(m))$, or
- $\exists e_c^{(k)} : (e_a^{(i)} \rightarrow e_c^{(k)}) \wedge (e_c^{(k)} \rightarrow e_b^{(j)})$.

For two local states s_1 and s_2 , $s_1 \rightarrow s_2$ iff the ending of s_1 happen-before (or coincides with) the beginning of s_2 (note that the beginning and ending of a state are both events). As shown in Fig. 1(a), $s_2^{(2)} \rightarrow s_1^{(1)}$ and $s_4^{(1)} \rightarrow s_5^{(2)}$.



(a) Sliding windows over asynchronous event streams



(b) The n -dimensional sliding window over the lattice

Fig. 1. System model

One *checker process* P_{che} is in charge of collecting and processing the asynchronous event streams. For example in a context-aware computing scenario [2], P_{che} may be a context reasoning process deployed over the context-aware middleware. In a supply chain management scenario, P_{che} may be a central administration application, monitoring the progresses of multiple supply chains.

Whenever $P^{(k)}$ generates a new event and proceeds to a new local state, it sends the local state with the vector clock timestamp to P_{che} . We use message sequence numbers to ensure that P_{che} receives messages from each $P^{(k)}$ in FIFO manner [15], [16], [1], [2].

2.2 Lattice of Consistent Global States(CGS)

In the tracking/monitoring application, we are concerned with the state of the entities being monitored after specific events are executed. For a system of asynchronous event streams, we are thus concerned with the global states or snapshots of the whole system.

A global state $\mathcal{G} = (s^{(1)}, s^{(2)}, \dots, s^{(n)})$ of asynchronous event streams is defined as a vector of local states from each non-checker process $P^{(k)}$. A global state may be either consistent or inconsistent. The notion of *Consistent Global State (CGS)* is crucial in processing of asynchronous

event streams. Intuitively, a global state is consistent if an omniscient external observer could actually observe that the system enters that state. Formally, a global state \mathcal{C} is *consistent* if and only if the constituent local states are pairwise concurrent [6], i.e.,

$$\mathcal{C} = (s^{(1)}, s^{(2)}, \dots, s^{(n)}), \forall i, j : i \neq j :: \neg(s^{(i)} \rightarrow s^{(j)})$$

The CGS denotes a global snapshot or meaningful observation of the system of asynchronous event streams.

It is intuitive to define the *precede* relation (denoted by ' \prec ') between two CGSs: $\mathcal{C} \prec \mathcal{C}'$ if \mathcal{C}' is obtained via advancing \mathcal{C} by exactly one local state on one non-checker process. The *lead-to* relation (denoted by ' \rightsquigarrow ') is defined as the transitive closure of ' \prec '.

The set of all CGSs together with the ' \rightsquigarrow ' relation form a distributive lattice [6], [7]. As shown in Fig. 1(b), black dots denote the CGSs and the edges between them depict the ' \prec ' relation. The crosses " \times " denote the inconsistent global states. The lattice structure among all CGSs serves as a key notion for the detection of global predicates over asynchronous event streams [6], [7].

2.3 The n -dimensional Sliding Window over Asynchronous Event Streams

On P_{che} , states of each event source $P^{(k)}$ are queued in $Que^{(k)}$. As discussed in Section 1, in many cases, it is too expensive and often unnecessary to process the entire event stream. A *local sliding window* $W^{(k)}$ of size w is imposed on each $Que^{(k)}$. Then we can define the *n -dimensional sliding window* W as the Cartesian product of each $W^{(k)}$: $W = W^{(1)} \times W^{(2)} \times \dots \times W^{(n)}$.

As shown in Fig. 1(a), the window $W^{(1)}$ with $w = 3$ on $P^{(1)}$ currently contains $\{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}\}$. The 2-dimensional sliding window $W^{(1)} \times W^{(2)}$ is depicted by the gray square in Fig. 1(b). The arrival of $s_5^{(1)}$ will trigger the 2-dimensional window to slide in $P^{(1)}$'s dimension, and $W^{(1)}$ is updated to $\{s_3^{(1)}, s_4^{(1)}, s_5^{(1)}\}$.

We assume that the concurrency control scheme is available on P_{che} , which means that the events from all non-checker processes are processed one at a time. We also assume that the sliding windows on the event streams have uniform size w . Note that this assumption is not restrictive and is for the ease of interpretation. Our proposed scheme also works if the windows on different streams have different sizes.

3 LAT-WIN - DESIGN OVERVIEW

The central problem in this work is how to characterize and maintain the n -dimensional sliding window over asynchronous event streams. Toward this problem, our contribution is two-fold. First, we characterize *Lat-Win* - the lattice of CGSs over the asynchronous event streams within the n -dimensional sliding window. Then we propose an online algorithm to maintain *Lat-Win* at runtime.

3.1 Characterization of Lat-Win

An important property concerning *Lat-Win* is that all CGSs within the n -dimensional sliding window together with the ' \rightsquigarrow ' relation have the lattice structure. Moreover, *Lat-Win* turns out

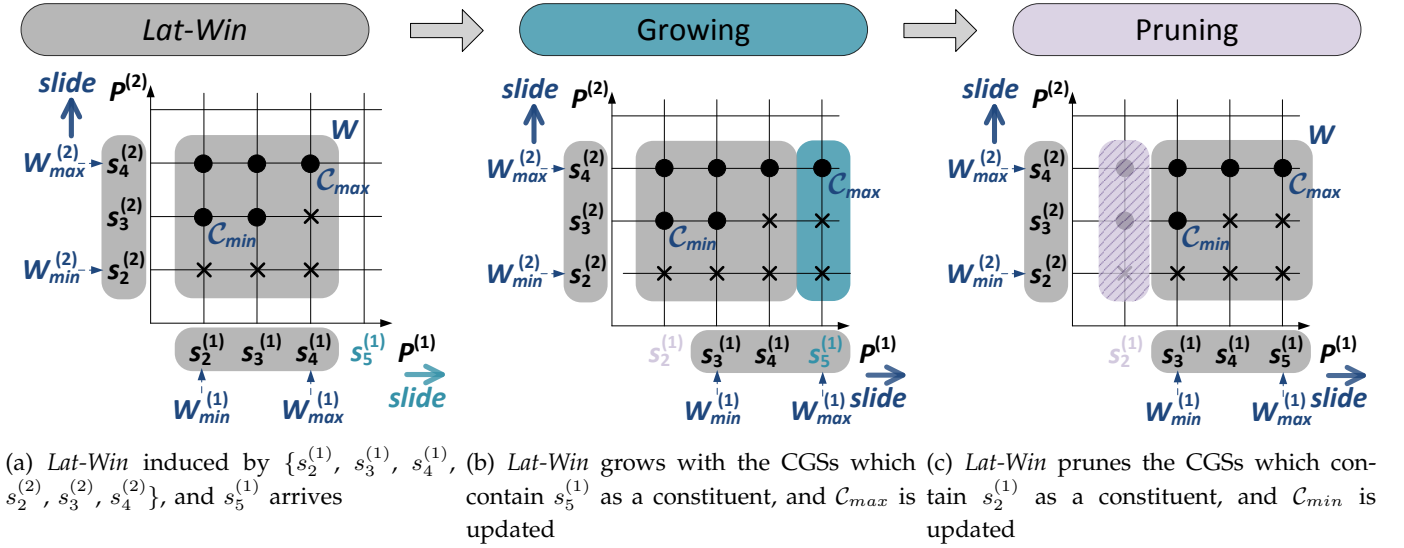


Fig. 2. Online maintenance of *Lat-Win*. When $s_5^{(1)}$ arrives, *Lat-Win* first grows with a set of new CGSs and then prunes the stale CGSs.

to be a distributive convex sublattice of the original lattice *LAT* (the lattice obtained when no sliding window is used and all event streams are processed). As shown in Fig. 1(b), the gray square in the middle is a 2-dimensional sliding window over two asynchronous event streams produced by $P^{(1)}$ and $P^{(2)}$. The CGSs within the square form a convex sublattice of the original lattice, i.e., the *Lat-Win*.

When an event $e_j^{(i)}$ is executed and $P^{(i)}$ arrives at a new local state $s_j^{(i)}$, the stale local state $s_k^{(i)}$ ($j - k = w$) in window $W^{(i)}$ will be discarded. The *Lat-Win* will “grow” with a set of CGSs consisting of $s_j^{(i)}$ and other local states from $W^{(m)}$ ($m \neq i$), and “prune” the CGSs which contain $s_k^{(i)}$ as a constituent.

For example in Fig. 2, assume that the *Lat-Win* is initially shown in Fig. 2(a). When a new local state $s_5^{(1)}$ arrives, $s_2^{(1)}$ will be discarded. State $s_5^{(1)}$ will be combined with local states in $W^{(2)}$ to obtain the CGS $C_{5,4} = (s_5^{(1)}, s_4^{(2)})$ in the blue rectangle in Fig. 2(b). CGSs which contain $s_2^{(1)}$ as a constituent in the left shaded rectangle will be discarded as shown in Fig. 2(c). The CGSs in the current window (e.g., the gray square in Fig. 2(c)) remain to be a sublattice. It seems that the 2-dimensional window containing the *Lat-Win* slides over the asynchronous event streams produced by $P^{(1)}$ and $P^{(2)}$.

3.2 Online Maintenance of *Lat-Win*

Based on the theoretical characterization above, we propose an algorithm for maintaining *Lat-Win* at runtime. Let C_{min} (C_{max}) denote the CGS which has no predecessors (successors) in *Lat-Win*. C_{min} and C_{max} serve as two “anchors” in updating *Lat-Win*. When a new local state arrives, the *Lat-Win* “grows” from C_{max} and “prunes” from C_{min} , as shown in Fig. 2. After the growing and pruning, C_{min} and C_{max} are also updated for further maintenance of *Lat-Win*. Due to the symmetry in the lattice structure, the growing and pruning of *Lat-Win* are dual. So are the updates of C_{min} and C_{max} .

4 LAT-WIN - CHARACTERIZING THE SNAPSHOTS OF WINDOWED ASYNCHRONOUS EVENT STREAMS

The theoretical characterization of *Lat-Win* consists of two parts. First we study the lattice of snapshots within the n -dimensional sliding window. Then we study how the *Lat-Win* evolves as the n -dimensional window slides.

4.1 Sub-lattice within the Sliding Window

An n -dimensional sliding window consists of n local windows sliding on event streams produced by non-checker processes $P^{(1)}, P^{(2)}, \dots, P^{(n)}$, and induces n segments of local states $W^{(1)}, W^{(2)}, \dots, W^{(n)}$.

The happen-before relation between local states has been encoded in their logical clock timestamps. Based on the local states as well as the happen-before relation among them, we can get a set of CGSs within the n -dimensional sliding window. An important property we find is that the CGSs within the n -dimensional sliding window, together with the ' \rightsquigarrow ' relation, also form a lattice - *Lat-Win*. More importantly, *Lat-Win* is a *distributive convex sub-lattice* of the original lattice *LAT*. Formally,

Theorem 1. *Given an n -dimensional sliding window $W = W^{(1)} \times W^{(2)} \times \dots \times W^{(n)}$ over asynchronous event streams, let $Set_C(W)$ denote the CGSs constructed from local states in W . If $Set_C(W)$ is not empty,*

1. *($Set_C(W), \rightsquigarrow$) forms a lattice, denoted by *Lat-Win*;*
2. **Lat-Win* is a sublattice of *LAT*;*
3. **Lat-Win* is convex and distributive.*

Proof:

1.1: A *lattice* is a poset L such that for all $x, y \in L$, the least upper bound (*join*) of x and y (denoted $x \sqcup y$) and the greatest lower bound (*meet*) of x and y (denoted $x \sqcap y$) exist and are contained in the poset. For two CGSs C_i, C_j , $C_i \sqcap C_j = (\min(C_i[1], C_j[1]), \dots, \min(C_i[n], C_j[n]))$, $C_i \sqcup C_j = (\max(C_i[1], C_j[1]), \dots, \max(C_i[n], C_j[n]))$.

We prove it by contradiction. Assume that $\exists C_i, C_j \in Set_C(W)$ and $C_i \sqcap C_j$ does not exist. It is obvious that $C_i \sqcap C_j$ is unique, so as $C_i \sqcup C_j$. It is to say that $C_i \sqcap C_j$ is not a CGS, that is, $\exists s, t, \min(C_i[s], C_j[s]) \rightarrow \min(C_i[t], C_j[t])$. Assume without loss of generality that $\min(C_i[s], C_j[s]) = C_i[s]$, i.e., $C_i[s] \rightarrow C_j[s]$ or $C_i[s] = C_j[s]$. Then, we get $C_i[s] \rightarrow \min(C_i[t], C_j[t])$. Thus, $C_i[s] \rightarrow C_i[t]$, which is contrary to that C_i is a CGS. Thus, $C_i \sqcap C_j$ exists. The proof of the existence of $C_i \sqcup C_j$ is the same.

It is easy to prove that $C_i \sqcap C_j$ and $C_i \sqcup C_j$ are both in $Set_C(W)$, because the constituent local states of $C_i \sqcap C_j$ and $C_i \sqcup C_j$ are all in $W^{(1)}, W^{(2)}, \dots, W^{(n)}$, and $Set_C(W)$ contains all the CGSs constructed from local states in $W^{(1)}, W^{(2)}, \dots, W^{(n)}$. Thus, $(Set_C(W), \rightsquigarrow)$ forms a lattice.

1.2: Let $Set_C(LAT)$ denote the CGSs of the original lattice *LAT*. A subset $S \subseteq L$, is a *sublattice* of lattice L , iff S is non-empty and $\forall a, b \in S, ((a \sqcap b) \in S) \wedge ((a \sqcup b) \in S)$. It is obvious that $Set_C(W)$ of *Lat-Win* is a subset of $Set_C(LAT)$. From the proof of Theorem 1.1, we can easily prove that *Lat-Win* is a sublattice of *LAT*.

1.3: A subset S of a lattice L is called *convex* iff $\forall a, b \in S, c \in L$, and $a \leq c \leq b$ imply that $c \in S$ (see Section I.3 in [17]). For three CGSs $C_i, C_j \in \text{Set}_C(W)$, $C_k \in \text{Set}_C(LAT)$, $C_i \rightsquigarrow C_k \rightsquigarrow C_j$, it infers that $\forall t, (C_i[t] \rightarrow C_k[t] \text{ or } C_i[t] = C_k[t]) \wedge (C_k[t] \rightarrow C_j[t] \text{ or } C_k[t] = C_j[t])$. Note that $C_i[t], C_j[t] \in W^{(t)}$ and $W^{(t)}$ contains all local states within $[C_i[t], C_j[t]]$. Thus, $C_k[t] \in W^{(t)}$, and $C_k \in \text{Set}_C(W)$. Thus, *Lat-Win* is a convex sublattice of the original lattice *LAT*.

It is a well known result in lattice theory [18] that the set of all CGSs of a distributed computation forms a distributive lattice under the \subseteq relation. Thus, *LAT* is a distributive lattice. It can be proved that any sublattice of a distributive lattice is also a distributive lattice [18]. Thus, *Lat-Win* is also a distributive lattice. \square

The geometric interpretation of Theorem 1 is that W can be viewed as an n -dimensional “cube” over the original lattice, and CGSs within this cube also form a lattice *Lat-Win*. Moreover, the ‘convex’ and ‘distributive’ properties of the original lattice *LAT* preserve when we focus on CGSs within the cube. Let $C_{i,j} = (s_i^{(1)}, s_j^{(2)})$. As shown in Fig. 2(a), the local windows are $W^{(1)} = \{s_2^{(1)}, s_3^{(1)}, s_4^{(1)}\}$ and $W^{(2)} = \{s_2^{(2)}, s_3^{(2)}, s_4^{(2)}\}$. They define a square on the original lattice (Fig. 1(b)) and induce a sublattice $\text{Lat-Win} = (\{C_{2,3}, C_{2,4}, C_{3,3}, C_{3,4}, C_{4,4}\}, \rightsquigarrow)$. The induced *Lat-Win* is convex because all CGSs “greater than” $C_{2,3}$ and “smaller than” $C_{4,4}$ in the original lattice are contained in the *Lat-Win*.

Given *Lat-Win* defined in Theorem 1, we further study how *Lat-Win* is contained in the cube. Is this cube a tight wrapper, i.e., does *Lat-Win* span to the boundary of the cube? First note that the maximal CGS and the minimal CGS are both important to the update of *Lat-Win*. Intuitively, the maximal CGS C_{max} of *Lat-Win* is on the upper bound $W_{max}^{(i)}$ of at least one local window $W^{(i)}$, so that *Lat-Win* could grow with newly arrived local states from $P^{(i)}$. Dually, the minimal CGS C_{min} of the *Lat-Win* is on the lower bound $W_{min}^{(j)}$ of at least one local window $W^{(j)}$, so that *Lat-Win* could grow from the stale local states from $P^{(j)}$ in the past. Formally,

Theorem 2. *If Lat-Win is not empty,*

1. $\exists i, C_{max}[i] = W_{max}^{(i)}$;
2. $\exists j, C_{min}[j] = W_{min}^{(j)}$.

Proof:

2.1: Let $S_{succ}(s_j^{(i)})$ ($E_{succ}(s_j^{(i)})$) denote the successor local state (event) to local state $s_j^{(i)}$ on $P^{(i)}$, i.e., $S_{succ}(s_j^{(i)}) = s_{j+1}^{(i)}$, $E_{succ}(s_j^{(i)}) = e_{j+1}^{(i)}$. Let $sub(\mathcal{G}, i)$ denote the global state formed by combining global state \mathcal{G} and $S_{succ}(\mathcal{G}[i])$ (i.e., $sub(\mathcal{G}, i)[i] = S_{succ}(\mathcal{G}[i])$, $\forall j \neq i, sub(\mathcal{G}, i)[j] = \mathcal{G}[j]$).

We prove it by contradiction. If *Lat-Win* is not empty and $\forall i, C_{max}[i] \neq W_{max}^{(i)}$, then $\forall i, \exists S_{succ}(C_{max}[i]) \in W^{(i)}$. Because C_{max} is the maximal CGS, $\forall i$, global state $sub(C_{max}, i)$ is not CGS.

Global state $sub(C_{max}, i)$ is not CGS, $\exists j \notin \{i\}, C_{max}[j] \rightarrow S_{succ}(C_{max}[i])$ and $E_{succ}(C_{max}[j]) \rightarrow E_{succ}(C_{max}[i])$. Global state $sub(C_{max}, j)$ is not CGS, $\exists k \notin \{i, j\}, C_{max}[k] \rightarrow S_{succ}(C_{max}[j])$ and $E_{succ}(C_{max}[k]) \rightarrow E_{succ}(C_{max}[j])$. (If $k \in \{i, j\}$, we can get that $E_{succ}(C_{max}[i]) \rightarrow E_{succ}(C_{max}[j]) \rightarrow E_{succ}(C_{max}[i])$ or $E_{succ}(C_{max}[j]) \rightarrow E_{succ}(C_{max}[j])$, which is contrary to irreflexivity). By induction on the length of the set containing the used indexes ($\{i, j\}$ above), we can get that to the last

global state $sub(\mathcal{C}_{max}, m)$, the set contains all the indexes, and $\forall t \in \{1, 2, \dots, n\}, E_{succ}(\mathcal{C}_{max}[t]) \rightarrow E_{succ}(\mathcal{C}_{max}[m])$ (If $t \in \{1, 2, \dots, n\}$, it will lead to the contradiction to irreflexivity). Thus, if $LatWin$ is not empty, $\exists i, \mathcal{C}_{max}[i] = W_{max}^{(i)}$.

2.2: The proof is dual as above. \square

As shown in Fig. 2(a), the maximal CGS $\mathcal{C}_{4,4}[1] = s_4^{(1)} = W_{max}^{(1)}$, $\mathcal{C}_{4,4}[2] = s_4^{(2)} = W_{max}^{(2)}$ and the minimal CGS $\mathcal{C}_{2,3}[1] = s_2^{(1)} = W_{min}^{(1)}$.

4.2 Update of Lat-Win when the Window Slides

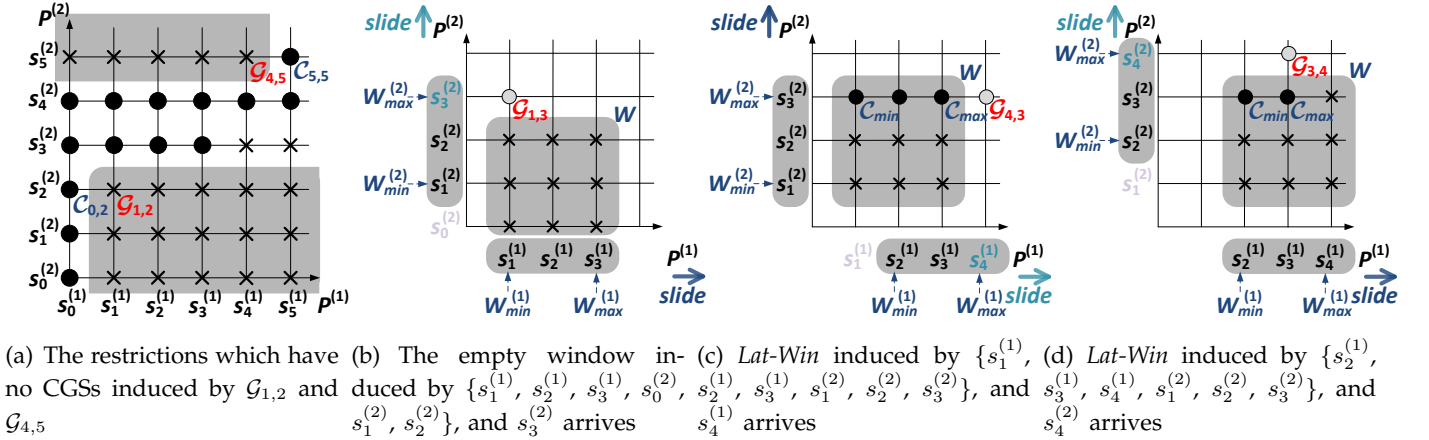


Fig. 3. Restrictions and the slide of the n -dimensional window. Assume the arrival of local states is $s_0^{(1)}, s_1^{(1)}, s_2^{(1)}, s_0^{(2)}, s_1^{(2)}, s_2^{(2)}, s_3^{(1)}, s_3^{(2)}, s_4^{(1)}, s_4^{(2)}, \dots$.

In this section, we discuss the update of *Lat-Win* when the n -dimensional window slides. Informally, the window slides as a new event is executed on $P^{(k)}$ and $P^{(k)}$ arrives at a new local state.

When a new local state from $P^{(k)}$ arrives, the stale local state (i.e., the old $W_{min}^{(k)}$) will be discarded. *Lat-Win* will grow with the CGSs containing the newly arrived local state, and prune the CGSs containing the stale local state, as shown in Fig. 2. Since the intersection between the set of new CGSs and the set of stale CGSs is empty, the growing and pruning of *Lat-Win* can be proceeded in any order. In this work, we first add newly obtained CGSs to *Lat-Win* and then prune the stale CGSs. During the growing and pruning process, \mathcal{C}_{min} and \mathcal{C}_{max} are also updated for further updates of *Lat-Win*.

We characterize the evolution of *Lat-Win* in three steps:

- Lemma 3 defines the *restrictions* of lattice, which serves as the basis for further growing and pruning;
- Theorem 4 defines the condition when *Lat-Win* can grow and Theorem 5 identifies the new \mathcal{C}_{min} and \mathcal{C}_{max} when *Lat-Win* grows;
- Theorem 6 defines the condition when *Lat-Win* can prune and Theorem 7 identifies the new \mathcal{C}_{min} and \mathcal{C}_{max} when *Lat-Win* prunes.

The growing and pruning are dual, as well as the updates of \mathcal{C}_{min} and \mathcal{C}_{max} .

4.2.1 Restrictions

Before we discuss the update of *Lat-Win*, we first introduce the notion of *restrictions*. When we obtain a global state and decide that it is not consistent, we can induce a specific region containing only inconsistent global states. The specific regions are also called *restrictions* in [9].

The geometric interpretation can be illustrated by the example in Fig. 3(a), global states $\mathcal{G}_{1,2} = (s_1^{(1)}, s_2^{(2)})$ and $\mathcal{G}_{4,5} = (s_4^{(1)}, s_5^{(2)})$ are not consistent ($s_2^{(2)} \rightarrow s_1^{(1)}$ and $s_4^{(1)} \rightarrow s_5^{(2)}$ in Fig. 1(a)). When looking from $\mathcal{C}_{0,2}$, $\mathcal{G}_{1,2}$ makes the lower gray region have no CGSs. When looking from $\mathcal{C}_{5,5}$, $\mathcal{G}_{4,5}$ makes the upper gray region have no CGSs. Formally, we have the following lemma:

Lemma 3. To a CGS \mathcal{C} of a lattice, and two global states $\mathcal{G}_1, \mathcal{G}_2$, $\mathcal{G}_1[i]$ ($\mathcal{G}_2[i]$) is the first (last) local state after (before) $\mathcal{C}[i]$ on $P^{(i)}$, $\forall k \neq i$, $\mathcal{G}_1[k] = \mathcal{G}_2[k] = \mathcal{C}[k]$,

1. If \mathcal{G}_1 is not CGS, then $\exists j \neq i, \mathcal{G}_1[j] \rightarrow \mathcal{G}_1[i]$, and none of the global states in the following set is CGS: $\{ \mathcal{G} | \mathcal{G}[j] \rightarrow \mathcal{G}_1[j] \text{ or } \mathcal{G}[j] = \mathcal{G}_1[j], \mathcal{G}_1[i] \rightarrow \mathcal{G}[i] \text{ or } \mathcal{G}_1[i] = \mathcal{G}[i] \}$;
2. If \mathcal{G}_2 is not CGS, then $\exists j \neq i, \mathcal{G}_2[i] \rightarrow \mathcal{G}_2[j]$, and none of the global states in the following set is CGS: $\{ \mathcal{G} | \mathcal{G}_2[j] \rightarrow \mathcal{G}[j] \text{ or } \mathcal{G}_2[j] = \mathcal{G}[j], \mathcal{G}[i] \rightarrow \mathcal{G}_2[i] \text{ or } \mathcal{G}[i] = \mathcal{G}_2[i] \}$.

Proof:

3.1: If \mathcal{G}_1 is not CGS, it is easy to verify that $\exists j \neq i, \mathcal{G}_1[j] \rightarrow \mathcal{G}_1[i]$. To any global state \mathcal{G} in $\{ \mathcal{G} | \mathcal{G}[j] \rightarrow \mathcal{G}_1[j] \text{ or } \mathcal{G}[j] = \mathcal{G}_1[j], \mathcal{G}_1[i] \rightarrow \mathcal{G}[i] \text{ or } \mathcal{G}_1[i] = \mathcal{G}[i] \}$, it is easy to verify that $\mathcal{G}[j] \rightarrow \mathcal{G}[i]$. Thus, global state \mathcal{G} is not consistent, and none of the global states in the set is CGS.

3.2: The proof is dual as above. □

4.2.2 Growing of Lat-Win

On the arrival of a new local state $s_i^{(k)}$, the n -dimensional window slides in $P^{(k)}$'s dimension, i.e., $W_{max}^{(k)} = s_i^{(k)}$, and a set of newly obtained CGSs (containing $s_i^{(k)}$) will be added into *Lat-Win*. We find that the growing process does not have to explore the whole combinational space of the new local state with all local states from every $W^{(k)}$. If *Lat-Win* is not empty, it will grow from \mathcal{C}_{max} in *Lat-Win*. The reason is that, if the next global state growing from \mathcal{C}_{max} is not consistent, as \mathcal{G}_1 in Lemma 3, it can be proved that the global states containing the newly arrived local state as a constituent are all in the restriction induced by a further global state and therefore not consistent. When *Lat-Win* is empty, the lattice can grow iff one CGS can be obtained containing the new local state and a lower bound of some local window. This is because as discussed in Theorem 2, the new \mathcal{C}_{min} should contain at least a lower bound of a local window. Formally,

Theorem 4. When a new event $e_i^{(k)}$ is executed on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives,

1. If *Lat-Win* $\neq \emptyset$, then *Lat-Win* can grow iff $\mathcal{C}_{max}[k] = s_{i-1}^{(k)}$ (the old $W_{max}^{(k)}$) and global state \mathcal{G} ($\mathcal{G}[k] = s_i^{(k)}, \forall j \neq k, \mathcal{G}[j] = \mathcal{C}_{max}[j]$) is CGS;
2. If *Lat-Win* $= \emptyset$, then *Lat-Win* can grow iff $\{ \mathcal{C} | \mathcal{C}[k] = s_i^{(k)}, \exists j \neq k, \mathcal{C}[j] = W_{min}^{(j)}, \mathcal{C} \text{ is CGS} \} \neq \emptyset$.

Proof:

4.1: “ \Rightarrow ”: We first prove that if *Lat-Win* is not empty and can grow, $\mathcal{C}_{max}[k] = s_{i-1}^{(k)}$. We prove it by contradiction. If $\mathcal{C}_{max}[k] \neq s_{i-1}^{(k)}$, $\mathcal{C}_{max}[k] \rightarrow s_{i-1}^{(k)}$ and $\exists S_{succ}(\mathcal{C}_{max}[k]) \in W^{(i)}$. From the proof of Theorem 2.1, we can easily verify that $\exists j \neq k, \mathcal{C}_{max}[j] = W_{max}^{(j)}$ and $W_{max}^{(j)} \rightarrow S_{succ}(\mathcal{C}_{max}[k])$. Note that $S_{succ}(\mathcal{C}_{max}[k]) \rightarrow s_i^{(k)}$. Thus, to any new global states \mathcal{G}' containing $s_i^{(k)}$, $\mathcal{G}'[j] \rightarrow \mathcal{G}'[k]$. By Lemma 3, *Lat-Win* cannot grow, which is contract to that *Lat-Win* can grow. Thus, if *Lat-Win* is not empty and can grow, $\mathcal{C}_{max}[k] = s_{i-1}^{(k)}$.

We then prove that if *Lat-Win* is not empty and can grow, global state \mathcal{G} ($\mathcal{G}[k] = s_i^{(k)}$, $\forall j \neq k, \mathcal{G}[j] = \mathcal{C}_{max}[j]$) is CGS. We prove it by contradiction. If \mathcal{G} is not CGS, $\exists j \neq k, \mathcal{G}[j] \rightarrow \mathcal{G}[k]$, $E_{succ}(\mathcal{C}_{max}[j]) \rightarrow E_{succ}(\mathcal{C}_{max}[k])$ ($\mathcal{G}[j] = \mathcal{C}_{max}[j], \mathcal{G}[k] = succ(\mathcal{C}_{max}[k])$). If $\mathcal{C}_{max}[j] \neq W_{max}^{(j)}$, by the proof of Theorem 2.1, we can easily verify that $\exists m \notin \{j, k\}, \mathcal{C}_{max}[m] = W_{max}^{(m)}$, $W_{max}^{(m)} \rightarrow S_{succ}(\mathcal{C}_{max}[j])$ and $E_{succ}(W_{max}^{(m)}) \rightarrow E_{succ}(\mathcal{C}_{max}[j])$. Thus, $E_{succ}(W_{max}^{(m)}) \rightarrow E_{succ}(\mathcal{C}_{max}[k])$. Thus, to any new global states \mathcal{G}' containing $s_i^{(k)}$, $\mathcal{G}'[m] \rightarrow \mathcal{G}'[k]$. By Lemma 3, *Lat-Win* cannot grow. Thus, if *Lat-Win* is not empty and can grow, global state \mathcal{G} ($\mathcal{G}[k] = s_i^{(k)}$, $\forall j \neq k, \mathcal{G}[j] = \mathcal{C}_{max}[j]$) is CGS.

“ \Leftarrow ”: Global state \mathcal{G} is CGS, thus *Lat-Win* can grow.

4.2: “ \Rightarrow ”: It is easy to verify the theorem by Theorem 2.2. It can be proved by contradiction. If $\{\mathcal{C} | \mathcal{C}[k] = s_i^{(k)}, \exists j \neq k, \mathcal{C}[j] = W_{min}^{(j)}, \mathcal{C} \text{ is CGS}\} = \emptyset$, after the process of growing, the new \mathcal{C}_{min} contains $s_i^{(k)}$, and $\forall j \neq k, W_{min}^{(j)} \rightarrow \mathcal{C}_{min}[j]$. By combining the predecessor local state of each $\mathcal{C}_{min}[j]$ with \mathcal{C}_{min} , a violation of irreflexivity will be inferred, dual as the proof of Theorem 2.1.

“ \Leftarrow ”: $\{\mathcal{C} | \mathcal{C}[k] = s_i^{(k)}, \exists j \neq k, \mathcal{C}[j] = W_{min}^{(j)}, \mathcal{C} \text{ is CGS}\} \neq \emptyset$, then *Lat-Win* can grow. \square

We illustrate the theorem by three examples in Fig. 3(b), Fig. 3(c), and Fig. 3(d), on the arrival of $s_3^{(2)}$, $s_4^{(1)}$, and $s_4^{(2)}$, respectively. In Fig. 3(b), the current *Lat-Win* is empty and $s_3^{(2)}$ arrives. The lattice can grow iff the global state $\mathcal{G}_{1,3} = (s_1^{(1)}, s_3^{(2)})$ is CGS. Note that $\mathcal{G}_{1,3}$ is CGS (as shown in Fig. 3(a)). Thus *Lat-Win* can grow to the new lattice in Fig. 3(c). In Fig. 3(c), the current *Lat-Win* is not empty and $s_4^{(1)}$ arrives. *Lat-Win* can grow iff $\mathcal{C}_{max}[1] = s_3^{(1)}$ and the global state $\mathcal{G}_{4,3} = (s_4^{(1)}, s_3^{(2)})$ is CGS. Note that $\mathcal{G}_{4,3}$ is not CGS (in Fig. 3(a)). Thus *Lat-Win* cannot grow, as shown in Fig. 3(d). In Fig. 3(d), the current *Lat-Win* is not empty and $s_4^{(2)}$ arrives. *Lat-Win* can grow iff $\mathcal{C}_{max}[2] = s_3^{(2)}$ and the global state $\mathcal{G}_{3,4} = (s_3^{(1)}, s_4^{(2)})$ is CGS. Note that $\mathcal{G}_{3,4}$ is CGS (in Fig. 3(a)). Thus *Lat-Win* can grow to the new lattice in Fig. 2(a).

The maximal and minimal CGSs are important to the update of *Lat-Win*. Thus, we discuss how to locate \mathcal{C}_{max} and \mathcal{C}_{min} after the growing of *Lat-Win* for further updates. After the growing of *Lat-Win*, the new \mathcal{C}_{max} should contain the new local state as a constituent. If *Lat-Win* was empty and grows with the new local state, \mathcal{C}_{min} should contain the new local state as a constituent.

For example in Fig. 3(b), *Lat-Win* is empty and can grow with the newly arrived local state $s_3^{(2)}$, the new $\mathcal{C}_{max}[2] = s_3^{(2)}$ and the new $\mathcal{C}_{min}[2] = s_3^{(2)}$, as shown in Fig. 3(c). Formally,

Theorem 5. When a new event $e_i^{(k)}$ is executed on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives,

1. If *Lat-Win* can grow, then $\mathcal{C}_{max}[k] = s_i^{(k)}$ (the new $W_{max}^{(k)}$); else \mathcal{C}_{max} remains.
2. If *Lat-Win* = \emptyset and can grow, then $\mathcal{C}_{min}[k] = s_i^{(k)}$; else \mathcal{C}_{min} remains.

Proof:

5.1: If *Lat-Win* can grow, all the new CGSs contain $s_i^{(k)}$ as a constituent. The CGS \mathcal{G} in Theorem 4.1 ensures that the new maximal CGS is at least “larger than” \mathcal{G} . Thus, the new \mathcal{C}_{max} is in the set of the new CGSs, and $\mathcal{C}_{max}[k] = s_i^{(k)}$.

5.2: If $\text{Lat-Win} = \emptyset$ and can grow, it is easy to verify $\mathcal{C}_{min}[k] = s_i^{(k)}$. \square

4.2.3 Pruning of Lat-Win

On the arrival of a new local state, after the growing of new CGSs, *Lat-Win* will prune the CGSs which contain the stale local state. The pruning does not have to explore the whole lattice to check whether a CGS contains the stale local state. Intuitively, *Lat-Win* can prune, iff *Lat-Win* is not empty and \mathcal{C}_{min} contains the stale local state. Formally,

Theorem 6. When a new event $e_i^{(k)}$ is executed on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives, after the growing, *Lat-Win* can prune, iff $\text{Lat-Win} \neq \emptyset$ and $\mathcal{C}_{min}[k] \rightarrow W_{min}^{(k)}$.

Proof:

“ \Rightarrow ”: If *Lat-Win* can prune, that is, there is at least a CGS containing the stale local state (the old $W_{min}^{(k)}$). Thus, $\mathcal{C}_{min}[k]$ equals the old $W_{min}^{(k)}$, and $\mathcal{C}_{min}[k] \rightarrow W_{min}^{(k)}$.

“ \Leftarrow ”: If $\text{Lat-Win} \neq \emptyset$ and $\mathcal{C}_{min}[k] \rightarrow W_{min}^{(k)}$, $\mathcal{C}_{min}[k]$ contains the stale local state. Thus, *Lat-Win* can prune. \square

For example, in Fig. 3(c), on the arrival of $s_4^{(1)}$, $\mathcal{C}_{min}[1] = s_1^{(1)}$ and $\mathcal{C}_{min}[1] \rightarrow W_{min}^{(1)}$. Thus *Lat-Win* can prune, as shown in Fig. 3(d). In Fig. 3(d), on the arrival of $s_4^{(2)}$, $\mathcal{C}_{min}[2] = s_3^{(2)}$ and $\mathcal{C}_{min}[2] \not\rightarrow W_{min}^{(2)}$. Thus *Lat-Win* does not have to prune, as shown in Fig. 2(a).

We then discuss how to locate \mathcal{C}_{max} and \mathcal{C}_{min} after the pruning of *Lat-Win* for further updates. When a new local state from $P^{(k)}$ arrives, after the pruning of *Lat-Win*, if *Lat-Win* prunes to be empty, the maximal and minimal CGSs are *null*. If *Lat-Win* prunes to be non-empty, the minimal CGS should contain the new $W_{min}^{(k)}$. Formally,

Theorem 7. When a new event $e_i^{(k)}$ is executed on $P^{(k)}$ and the new local state $s_i^{(k)}$ from $P^{(k)}$ arrives, after the growing,

1. If $\mathcal{C}_{max}[k] \rightarrow W_{min}^{(k)}$, then $\mathcal{C}_{max} = \text{null}$; else \mathcal{C}_{max} remains.
2. If $\mathcal{C}_{max}[k] \rightarrow W_{min}^{(k)}$, then $\mathcal{C}_{min} = \text{null}$; else if *Lat-Win* can prune, then $\mathcal{C}_{min}[k] = W_{min}^{(k)}$; else \mathcal{C}_{min} remains.

Proof:

7.1: If $\mathcal{C}_{max}[k] \rightarrow W_{min}^{(k)}$, all CGSs in *Lat-Win* contain the stale local state. Thus, *Lat-Win* prunes to be empty, and $\mathcal{C}_{max} = \text{null}$.

7.2: If $\mathcal{C}_{max}[k] \rightarrow W_{min}^{(k)}$, *Lat-Win* prunes to be empty, and $\mathcal{C}_{min} = \text{null}$; If $\mathcal{C}_{max}[k] \not\rightarrow W_{min}^{(k)}$ and *Lat-Win* can prune, there is at least a CGS containing $W_{min}^{(k)}$. Thus, the new $\mathcal{C}_{min}[k] = W_{min}^{(k)}$. \square

For example, in Fig. 3(c), on the arrival of $s_4^{(1)}$, *Lat-Win* can prune and $\mathcal{C}_{max}[1] \not\rightarrow W_{min}^{(1)}$, then the new $\mathcal{C}_{min}[1] = W_{min}^{(1)}$ and \mathcal{C}_{max} is not changed, as shown in Fig. 3(d).

5 LAT-WIN - ONLINE MAINTENANCE ALGORITHM

In this section, we present the design of the *Lat-Win* maintenance algorithm, based on the theoretical characterization above. \mathcal{C}_{min} and \mathcal{C}_{max} serve as two anchors in maintaining *Lat-Win*. When a new local state arrives, *Lat-Win* grows from \mathcal{C}_{max} and prunes from \mathcal{C}_{min} . After the growing and pruning, \mathcal{C}_{min} and \mathcal{C}_{max} are also updated for further maintenance of *Lat-Win*.

P_{che} is in charge of collecting and processing the local states sent from non-checker processes. Upon initialization, P_{che} gets the window size w and initializes n local windows $W^{(k)}$ over local states from each $P^{(k)}$. Upon receiving a new local state from $P^{(k)}$, P_{che} first enqueues the local state into $Que^{(k)}$ and then updates *Lat-Win* in the order of growing and pruning. Pseudo codes of the maintenance algorithm are listed in Algorithm 1.

Algorithm 1: *Lat-Win* maintenance algorithm

```

1 Upon Initialization
2 get window size  $w$ ;
3 initialize window buffers  $W^{(k)}$ ;
4 Upon Receiving local state  $(s_i^{(k)})$  from  $P^{(k)}$ 
5  $Que^{(k)}.enqueue(s_i^{(k)})$ ;
6 if  $s_i^{(k)} = Que^{(k)}.head()$  then
7   pop the front continuous local states of  $Que^{(k)}$  to the end of  $InputQue$ ;
8   trigger  $update()$ ;

9  $update()$ 
10 while  $InputQue \neq \emptyset$  do
11   pop  $s_i^{(k)} = InputQue.head()$ ;
12   push  $s_i^{(k)}$  into  $W^{(k)}$ ;                                /*  $W_{max}^{(k)} = s_i^{(k)}$  */
13   grow_lattice( $s_i^{(k)}, k$ );                                /* Algorithm 2 */
14   prune_lattice( $\mathcal{C}_{min}, k$ );                               /* Algorithm 3 */

```

5.1 Growing of *Lat-Win*

On the arrival of a new local state, the process of growing consists of three steps. First, it is checked whether *Lat-Win* can grow, as discussed in Theorem 4. If yes, *Lat-Win* will grow with a set of new CGSs containing the new arrived local state. During the step of growing, \mathcal{C}_{max} and \mathcal{C}_{min} are updated too, as discussed in Theorem 5. Pseudo codes of growing are listed in Algorithm 2.

When *Lat-Win* is empty, the lattice can grow iff the set S in line 2 contains a CGS, as discussed in Theorem 4.2. If S has a CGS, the $grow()$ sub-routine will be triggered to add the new CGSs. When *Lat-Win* is not empty, the lattice can grow iff \mathcal{C}_{max} and the next global state satisfy the condition defined in Theorem 4.1, as shown in line 5-8. If the condition is satisfied, the $grow()$ sub-routine will be triggered to add the new CGSs. The growing of *Lat-Win* is achieved by

recursively adding all the predecessors and successors of a CGS, as shown in line 10-19. During the growing process, \mathcal{C}_{max} and \mathcal{C}_{min} are also updated. Theorem 5 ensures that \mathcal{C}_{max} can be found in the new added CGSs, and that when *Lat-Win* was empty, \mathcal{C}_{min} can be found in the new added CGSs, as shown in line 12-13.

Algorithm 2: $grow_lattice(s_i^{(k)}, k)$

```

1 if Lat-Win =  $\emptyset$  then
2    $S = \{\mathcal{C} | \mathcal{C}[k] = s_i^{(k)}, \exists j \neq k, \mathcal{C}[j] = W_{min}^{(j)}, \mathcal{C} \text{ is CGS}\};$ 
3   if  $S \neq \emptyset$  then
4      $\lfloor$  get a CGS  $\mathcal{C}$  from  $S$ ;  $grow(\mathcal{C})$ ;
5 else if  $\mathcal{C}_{max}[k] = s_{i-1}^{(k)}$  then
6   combine  $\mathcal{C}_{max}$  and  $s_i^{(k)}$  to get a global state  $\mathcal{G}$ ;
7   if  $\mathcal{G}$  is CGS then
8      $\lfloor$  connect  $\mathcal{G}$  to  $\mathcal{C}_{max}$ ;  $grow(\mathcal{G})$ ;
9  $grow(\mathcal{C})$ 
10 Set  $prec(\mathcal{C}) = \{\mathcal{C}' | \forall i, \mathcal{C}'[i] \in W^{(i)}, \mathcal{C}' \text{ is CGS}, \mathcal{C}' \prec \mathcal{C}\};$ 
11 Set  $sub(\mathcal{C}) = \{\mathcal{C}' | \forall i, \mathcal{C}'[i] \in W^{(i)}, \mathcal{C}' \text{ is CGS}, \mathcal{C} \prec \mathcal{C}'\};$ 
12 if  $prec(\mathcal{C}) = \emptyset$  then  $\mathcal{C}_{min} = \mathcal{C}$ ;
13 if  $sub(\mathcal{C}) = \emptyset$  then  $\mathcal{C}_{max} = \mathcal{C}$ ;
14 foreach  $\mathcal{C}'$  in  $prec(\mathcal{C})$  do
15   if  $\mathcal{C}'$  does not exist then
16      $\lfloor$  connect  $\mathcal{C}'$  to  $\mathcal{C}$ ;  $grow(\mathcal{C}')$ ;
17 foreach  $\mathcal{C}'$  in  $sub(\mathcal{C})$  do
18   if  $\mathcal{C}'$  does not exist then
19      $\lfloor$  connect  $\mathcal{C}$  to  $\mathcal{C}'$ ;  $grow(\mathcal{C}')$ ;

```

5.2 Pruning of *Lat-Win*

Dually, the process of pruning consists of three steps as well. First, it is checked whether *Lat-Win* can prune, as discussed in Theorem 6. If yes, *Lat-Win* will prune the set of CGSs which contain the stale local state. During the step of pruning, \mathcal{C}_{max} and \mathcal{C}_{min} are updated too, as discussed in Theorem 7. Pseudo codes of pruning are listed in Algorithm 3.

The lattice can prune iff the condition in line 1 is satisfied, as discussed in Theorem 6. If $\mathcal{C}_{max}[k]$ is the stale local state ($\mathcal{C}[k]$ in line 2), *Lat-Win* will prune to be empty. Otherwise, the CGSs which contain the stale local state will be deleted, as shown in line 5-16. During the pruning process, \mathcal{C}_{min} is also updated. Theorem 7 ensures that \mathcal{C}_{min} can be found in the CGSs which contain the new $W_{min}^{(k)}$, as shown in line 13-15.

Algorithm 3: *prune_lattice*(\mathcal{C}, k)

```

1 if  $Lat-Win \neq \emptyset \ \&\& \ \mathcal{C}_{min}[k] \rightarrow W_{min}^{(k)}$  then
2   if  $\mathcal{C}_{max}[k] = \mathcal{C}[k]$  then
3      $Lat-Win = \emptyset, \mathcal{C}_{max} = \mathcal{C}_{min} = null;$ 
4   else
5     Set  $S = \{\mathcal{C}\};$ 
6     while  $S \neq \emptyset$  do
7       pop  $\mathcal{C}'$  from  $S;$ 
8       Set  $sub(\mathcal{C}') = \{\mathcal{C}'' | \mathcal{C}' \prec \mathcal{C}'', \mathcal{C}'' \text{ is CGS}\};$ 
9       foreach  $\mathcal{C}''$  in  $sub(\mathcal{C}')$  do
10        delete the connection between  $\mathcal{C}'$  and  $\mathcal{C}'';$ 
11        if  $\mathcal{C}''[k] = \mathcal{C}'[k] \ \&\& \ \mathcal{C}'' \notin S$  then
12          add  $\mathcal{C}''$  into  $S;$ 
13        else if  $\mathcal{C}''[k] = W_{min}^{(k)}$  then
14          Set  $prec(\mathcal{C}'') = \{\mathcal{C}''' | \mathcal{C}'' \prec \mathcal{C}''', \mathcal{C}''' \text{ is CGS}\};$           /* without  $\mathcal{C}'$  */
15          if  $prec(\mathcal{C}'') = \emptyset$  then  $\mathcal{C}_{min} = \mathcal{C}'';$ 
16        delete  $\mathcal{C}';$ 

```

5.3 Complexity Analysis

Regarding the space for storing a single CGS as one unit, the worst-case space cost of the original lattice maintenance is $O(p^n)$, where p is the upper bound of number of events of each non-checker process, and n is the number of non-checker processes. However, the worst-case space cost of sliding windows over the original lattice is bounded by the size w of the windows, that is, $O(w^n)$, where w is a fixed number and much less than p . Due to the incremental nature of Algorithm 2, the space cost of the incremental part of *Lat-Win* in each time of growing is $O(w^{n-1})$.

The worst-case time cost of growing (Algorithm 2) happens when all the global states in the blue rectangle in Fig. 2(b) are CGSs. Thus, the worst-case time of growing is $O(n^3 w^{n-1})$, where w^{n-1} is the number of the global states in the blue rectangle and n^3 is the time cost of checking whether a global state is consistent.

The worst-case time cost of pruning (Algorithm 3) happens when all the CGSs in the left shaded rectangle in Fig. 2(c) should be discarded. Thus, the worst-case time of pruning is $O(w^{n-1})$, where w^{n-1} is the worst-case number of the CGSs in the left shaded rectangle.

From the performance analysis we can see that, by tuning the sliding window size w , the cost of asynchronous event stream processing can be effectively bounded. This justifies the adoption of sliding windows when only recent part of the event streams are needed by the tracking/monitoring application, and the application needs to strictly bound the cost of event processing.

6 EXPERIMENTAL EVALUATION

In this section, we further investigate the performance of *Lat-Win* via a case study. We first describe a smart office scenario to demonstrate how our approach supports context-awareness in asynchronous pervasive computing scenarios. Then, we describe the experiment design. Finally, we discuss the evaluation results.

6.1 Achieving Context-awareness by On-line Processing of Asynchronous Event Streams

We simulate a smart office scenario, where a context-aware application on Bob's mobile phone automatically turns the phone to silent mode when Bob attends a lecture[2].

The application detects that Bob attends a lecture by specification of the concurrency property: C_1 : *location of Bob is the meeting room, a speaker is in the room, and a presentation is going on* [2]. The application needs to turn the phone to silent mode when the property definitely holds. Formally, $C_1 = Def(\phi_1 \wedge \phi_2 \wedge \phi_3)$, which is explained in detail below.

The location context is detected by Bob's smart phone. When the phone connects to the access point in the meeting room, we assume that Bob is in this room. Specifically, non-checker process $P^{(1)}$ is deployed on Bob's smart phone, which updates the phone's connection to access points. Local predicate $\phi_1 = \text{"the user's smart phone connects to the AP inside the meeting room"}$ is specified over $P^{(1)}$.

An RFID reader is deployed in the room to detect the speaker. Specifically, non-checker process $P^{(2)}$ is deployed on the RFID reader, and local predicate $\phi_2 = \text{"the RFID reader detects a speaker"}$ is specified over $P^{(2)}$.

We detect that a presentation is going on if the projector is working. Specifically, non-checker process $P^{(3)}$ is deployed over the projector, and local predicate $\phi_3 = \text{"the projector is on"}$ is specified over $P^{(3)}$.

Observe that the mobile phone, the RFID reader, and the projector do not necessarily have synchronized clocks, and Bob may not be willing to synchronize his mobile with other devices due to privacy concerns. They suffer from message delay of wireless communications. Furthermore, the recent data is more informative and useful to the context-aware application than stale data, thus the sliding window can be imposed over the streams of context.

Non-checker processes produce event streams and communications among them help establish the happen-before relation among events. A checker process P_{che} is deployed on the context-aware middleware to collect local states with logical clock timestamps from non-checker processes, and maintain *Lat-Win* at runtime. Based on *Lat-Win*, P_{che} can further detect the concurrency property C_1 [2] and notify the mobile phone to turn to silent mode.

The detection of concurrency properties assumes the availability of an underlying context-aware middleware. We have implemented the middleware based on one of our research projects - *Middleware Infrastructure for Predicate detection in Asynchronous environments* (MIPA) [13], [14], [2].

6.2 Experiment Design

The user's and speaker's stay inside and outside of the meeting room, as well as the working period of the projector are generated following the Poisson process. Specifically, the sensors

collect context data every 1 min. We model the message delay by exponential distribution. The average duration of local contextual activities is 25 mins and the interval between contextual activities is 5 mins. Lifetime of the experiment is up to 100 hours.

In the experiments, we first study the benefits of imposing sliding window over asynchronous event streams in the percentage of detection $Perc_{det}$ and the percentage of space cost $Perc_s$. $Perc_{det}$ is defined as the ratio of $\frac{N_{Lat-Win}}{N_{LAT}}$. Here, $N_{Lat-Win}$ denotes the number of times the algorithm detects the specified property on *Lat-Win*. N_{LAT} denotes the number of times the algorithm detects the specified property on the original lattice *LAT*. $Perc_s$ is defined as the ratio of $\frac{S_{Lat-Win}}{S_{LAT}}$. Here, $S_{Lat-Win}$ denotes the average size of *Lat-Win* as the window slides. S_{LAT} denotes the size of the original lattice over the lifetime of the experiment.

Then, we study the performance of *Lat-Win* in the probability of detection $Prob_{det}$, the space cost $S_{Lat-Win}$ and time cost $T_{Lat-Win}$. Here, $Prob_{det}$ is defined as the ratio of $\frac{N_{physical}}{N_{Lat-Win}}$. $N_{Lat-Win}$ is defined above and $N_{physical}$ denotes the number of times such property holds in the window when global time is available. $S_{Lat-Win}$ is defined above and $T_{Lat-Win}$ denotes the average time from the instant when P_{che} is triggered to that when the detection finishes.

6.3 Evaluation Results

In this section, we first discuss the benefits of imposing sliding window over asynchronous event streams. Then we discuss the performance of the *Lat-Win* maintenance algorithm.

6.3.1 Benefits of Sliding Window

In this experiment, we investigate the impact of window size w on the percentage of detection $Perc_{det}$ and the percentage of space cost $Perc_s$. We fix the average message delay to 0.5 s.

The experiment shows that the sliding window enables the trade-off between $Perc_{det}$ and $Perc_s$. As shown in Fig. 6.3.1, the increase of w results in monotonic increase in both $Perc_{det}$ and $Perc_s$. When we tune w from 1 to 4, $Perc_{det}$ (the upper blue line) increases quickly up to 97.11%. When we tune w from 4 to 10, $Perc_{det}$ increases slowly and remains quite high towards 100%. $Perc_s$ (the lower green line) increases almost linearly as w increases. When we set w to 10, $Perc_s$ remains small than 1%. We can find that, rather than maintaining the whole original lattice, imposing a quite small sliding window over asynchronous event streams can keep $Perc_{det}$ high and $Perc_s$ quite small. It indicates that recent data is more relevant and important to the application. When the window size is 4, the relative growth rate between $Perc_{det}$ and $Perc_s$ slows down, and $Perc_{det}$ is quite high, $S_{Lat-Win}$ is 27.23. Thus, in the following experiments, we set the window size w as 4 to study the performance of *Lat-Win*. (Notice that in different scenarios, the turning point of $w = 4$ may be different.)

6.3.2 Performance of *Lat-Win*

In this experiment, we investigate the performance of *Lat-Win* in the probability of detection $Prob_{det}$, the average space cost $S_{Lat-Win}$, and the average time cost $T_{Lat-Win}$. We first vary the asynchrony, i.e., the message delay. We also vary the window size w . Finally, we vary the number of non-checker processes n .

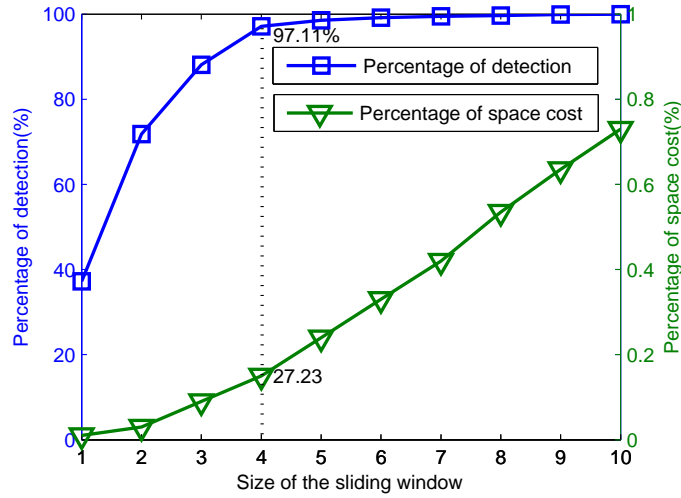


Fig. 4. Benefits of sliding window.

6.3.2.1 Effects of Tuning the Message Delay: In this experiment, we study how the message delay affects the performance of *Lat-Win*. We fix the window size w to 4. We find that when encountered with reasonably long message delay (less than 5 s), $Prob_{det}$ (the upper blue line) decreases slowly and remains over 85%, as shown in Fig. 6.3.2.1. When we tune the average message delay from 0 s to 5 s, $S_{Lat-Win}$ increases slowly to about 30, whereas the worst-case cost is $w^n = 64$, as discussed in Section 5.3. The decrease of $Prob_{det}$ and the increase of $S_{Lat-Win}$ are mainly due to the increase of the uncertainty caused by the asynchrony (i.e., the message delay). $N_{physical}$ is smaller than $N_{Lat-Win}$, because the detection algorithm may detect the property to be true but in physical world the property is not satisfied due to the increasing uncertainty caused by the asynchrony. Furthermore, the increase of message delay results in a slow increase in $T_{Lat-Win}$, which remains less than 1 ms.

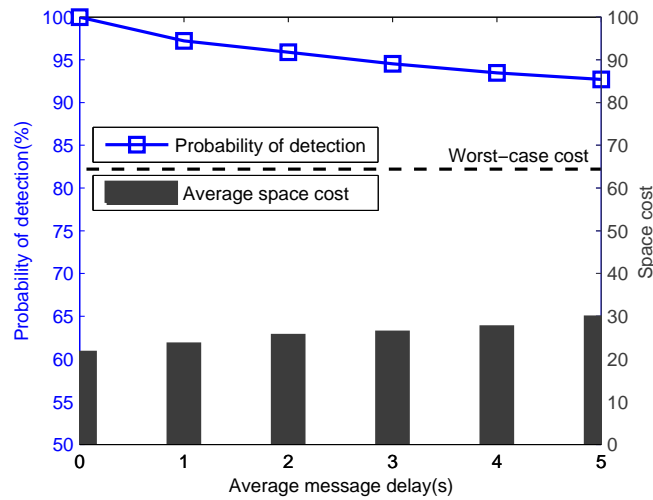


Fig. 5. Tuning the message delay.

6.3.2.2 Effects of Tuning Size of the Sliding Window: In this experiment, we study how the window size affects the performance of *Lat-Win*. We fix the message delay to 0.5 s. As shown

in Fig. 6.3.2.2, the increase of window size w results in monotonic increase in both $Prob_{det}$ and $S_{Lat-Win}$. When we tune w from 2 to 5, $Prob_{det}$ (the upper blue line) increases quickly up to 99%. The increase is mainly because that as w increases, the window has more information and thus can detect the property more accurately. $S_{Lat-Win}$ (the black bars) increases slowly as w increases, and is much smaller than the worst-case cost. Thus, imposing the sliding window over asynchronous event streams can achieve high accuracy while saving a large amount of space cost. Furthermore, the increase of window size results in a slow increase in $T_{Lat-Win}$, which remains less than 1 ms.

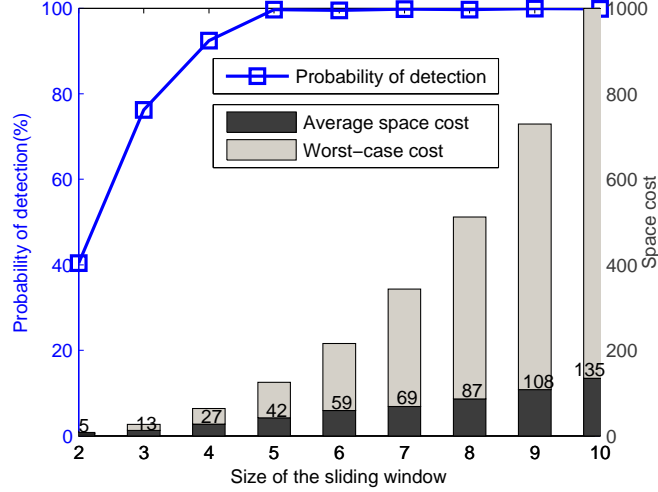


Fig. 6. Tuning size of the sliding window.

6.3.2.3 Effects of Tuning the Number of Non-checker Processes: In this experiment, we study how the number of non-checker processes n affects the performance of *Lat-Win*. We fix the average message delay to 0.5 s and the window size w to 4. We tune n from 2 to 9. As shown in Fig. 6.3.2.3, $Prob_{det}$ decreases linearly as n increases. When n increases from 2 to 9, $Prob_{det}$ decreases about 20%. The decrease is mainly because the asynchrony among non-checker processes accumulates as n increases. $S_{Lat-Win}$ increases exponentially as n increases. However, as n increases, the space cost of *Lat-Win* is much less than the worst-case (less than 1%), and even less than that of the original lattice. $S_{Lat-Win}$ and $T_{Lat-Win}$ are also shown in Table. 2. We find that $S_{Lat-Win}$ is approximately in $O((\theta w)^n)$, which is in accordance with the analysis in Section 5.3, where θ is a parameter associated with the asynchrony. In this experiment setting, θ is around 0.75. $T_{Lat-Win}$ also increases exponentially as $S_{Lat-Win}$ increases.

TABLE 2
Cost of Sliding Window When Tuning the Number of Non-checker Processes

Number of NPs (n)	2	3	4	5	6	7	8	9
$S_{Lat-Win}$	9	25	78	221	768	2691	9799	34408
$T_{Lat-Win}(ms)$	0.5	0.6	2.0	7.5	36	184	1616	14766

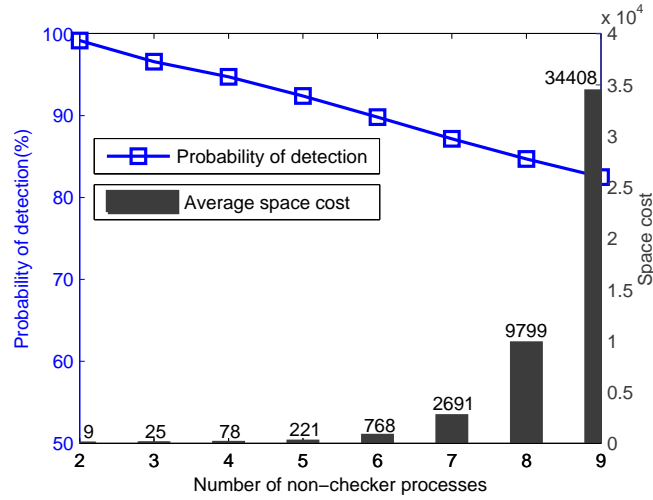


Fig. 7. Tuning the number of non-checker processes.

7 RELATED WORK

The lattice of global snapshots is a key notion in modeling behavior of asynchronous systems [7], [19], [20], [21], [22], [1], [23], [2], [24], [25], and is widely used in areas such as distributed program debugging [15], [16] and fault-tolerance [26]. One critical challenge is that the lattice of snapshots evolves to exponential size in the worst-case. Various schemes are used to cope with the lattice explosion problem [24], [25], [27], [10], [28]. For example, in [24], [25], the authors proposed the computation slice to efficiently compute all global states which satisfy a regular predicate[24]. In [10], the authors discussed that certain (useless) part of the lattice can be removed at runtime to reduce the size of lattice. In this work, we make use of the observation that, in many tracking/monitoring applications, it is often prohibitive and, more importantly, unnecessary to process the entire streams. Thus, we use sliding windows over distributed event streams to reduce the size of the lattice.

Sliding windows are widely used in event stream processing [29], [11], [12], [5]. Existing sliding windows are mainly designed over a single stream. However, it is not sufficiently discussed concerning the coordination of multiple (correlated) sliding windows over asynchronous event streams. We argue that the coordination of multiple windows is crucial to explicitly model and handle the asynchrony. In this work, to cope with the asynchrony among multiple event sources, we maintain a sliding window over each event stream. We consider the Cartesian product as an n -dimensional sliding window. Then we study the lattice of global snapshots of asynchronous event streams within the window.

8 CONCLUSION AND FUTURE WORK

In this work, we study the processing of asynchronous event streams within sliding windows. We first characterize the lattice structure of event stream snapshots within the sliding window. Then we propose an online algorithm to maintain *Lat-Win* at runtime. The *Lat-Win* is implemented and evaluated over MIPA.

In our future work, we will study how to make use of the partial asynchrony among event streams, to further improve the cost-effectiveness of event stream processing. We will also study the approximate/probabilistic detection of specified predicates over asynchronous event streams.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China (No. 60903024, 61021062) and the National 973 Program of China (2009CB320702).

REFERENCES

- [1] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu, "Concurrent event detection for asynchronous consistency checking of pervasive context," in *Proc. IEEE International Conference on Pervasive Computing and Communications (PERCOM'09)*, Galveston, Texas, USA, Mar. 2009.
- [2] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, "Runtime detection of the concurrency property in asynchronous pervasive computing environments," *IEEE Transactions on Parallel and Distributed Systems*, accepted in May 2011.
- [3] F. Niederman, R. Mathieu, R. Morley, and I. Kwon, "Examining rfid applications in supply chain management," *Communications of the ACM*, vol. 50, no. 7, pp. 92–101, 2007.
- [4] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 407–418.
- [5] S. Tirthapura, B. Xu, and C. Busch, "Sketching asynchronous streams over a sliding window," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, ser. PODC '06, 2006, pp. 82–91.
- [6] Özalp Babaoğlu and K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.
- [7] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: in search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, 1994.
- [8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [9] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. International Workshop on Parallel and Distributed Algorithms*, Holland, 1989, pp. 215–226.
- [10] C. Jard, G. Jourdan, T. Jeron, and J. Rampon, "A general approach to trace-checking in distributed computing systems," in *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE, 1994, pp. 396–403.
- [11] V. Braverman, R. Ostrovsky, and C. Zaniolo, "Optimal sampling from sliding windows," *Journal of Computer and System Sciences*, 2011.
- [12] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows: (extended abstract)," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '02, 2002, pp. 635–644.
- [13] "MIPA - Middleware Infrastructure for Predicate detection in Asynchronous environments." [Online]. Available: <http://mipa.googlecode.com>
- [14] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware support for context-awareness in asynchronous pervasive computing environments," in *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, dec. 2010, pp. 136–143.
- [15] V. K. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 299–307, Mar. 1994.
- [16] V. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 1323–1333, Dec. 1996.
- [17] G. A. Grätzer, *General Lattice Theory, 2nd Edition*. Birkhäuser, 2003.
- [18] B. Davey and H. Priestley, *Introduction to lattices and order, 2nd Edition*. Cambridge University Press, 2002.
- [19] Özalp Babaoğlu, E. Fromentin, and M. Raynal, "A unified framework for the specification and run-time detection of dynamic properties in distributed computations," *Journal of Systems and Software*, vol. 33, no. 3, pp. 287 – 298, 1996.
- [20] Özalp Babaoğlu and M. Raynal, "Specification and verification of dynamic properties in distributed computations," *Journal of Parallel and Distributed Computing*, vol. 28, no. 2, pp. 173 – 185, 1995.
- [21] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, New York, NY, USA, 1991, pp. 167–174.

- [22] T. Hua, Y. Huang, J. Cao, and X. Tao, "A lattice-theoretic approach to runtime property detection for pervasive context," in *Proceedings of the 7th international conference on Ubiquitous intelligence and computing*, ser. UIC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 307–321.
- [23] Y. Huang, J. Yu, J. Cao, and X. Tao, "Detection of behavioral contextual properties in asynchronous pervasive computing environments," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, dec. 2010, pp. 75–82.
- [24] N. Mittal, A. Sen, and V. K. Garg, "Solving computation slicing using predicate detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, pp. 1700–1713, December 2007.
- [25] A. Sen and V. Garg, "Formal verification of simulation traces using computation slicing," *IEEE Transactions on Computers*, pp. 511–527, 2007.
- [26] N. Mittal and V. K. Garg, "Techniques and applications of computation slicing," *Distributed Computing*, vol. 17, no. 3, pp. 251–277, 2005.
- [27] G. Dumais and H. F. Li, "Distributed predicate detection in series-parallel systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 373–387, April 2002.
- [28] L.-P. Chen, D.-J. Sun, and W. Chu, "Efficient online algorithm for identifying useless states in distributed systems," *Distributed Computing*, vol. 23, pp. 359–372, 2011.
- [29] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2002, pp. 1–16.